# Concurrent Binary Trees for Large-Scale Game Components

ANIS BENYOUB, Intel Corporation, France
JONATHAN DUPUY, Intel Corporation, France

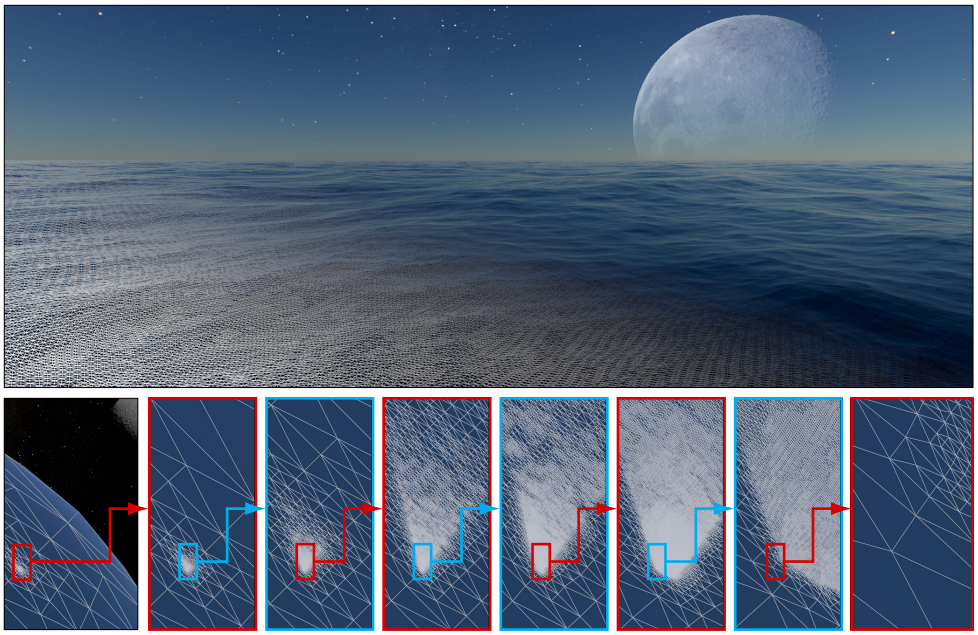


Fig. 1. (top) An Earth-sized planet rendered in real-time using our triangulation method. (bottom) An alternative wireframe-view of the render as seen from ground to space through successive zoomed-in insets.

A concurrent binary tree (CBT) is a GPU-friendly data-structure suitable for the generation of bisection based terrain tessellations, i.e., adaptive triangulations over square domains. In this paper, we expand the benefits of this data-structure in two respects. First, we show how to bring bisection based tessellations to arbitrary polygon meshes rather than just squares. Our approach consists of mapping a triangular subdivision primitive, which we refer to as a bisector, to each halfedge of the input mesh. These bisectors can then be subdivided adaptively to produce conforming triangulations solely based on halfedge operators. Second, we alleviate a limitation that restricted the triangulations to low subdivision levels. We do so by using the CBT as a memory pool manager rather than an implicit encoding of the triangulation as done originally. By using a CBT in this way, we concurrently allocate and/or release bisectors during adaptive subdivision using shared GPU memory.

Authors' Contact Information: Anis Benyoub, benyoub.anis@gmail.com, Intel Corporation, France; Jonathan Dupuy, etu.jdupuy@gmail.com, Intel Corporation, France.


ACM 2577-6193/2024/7-ART31
https://doi.org/10.1145/3675371

We demonstrate the benefits of our improvements by rendering planetary scale geometry out of very coarse meshes. Performance-wise, our triangulation method evaluates in less than 0.2ms on console-level hardware.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Massively parallel algorithms**.

Additional Key Words and Phrases: level-of-detail, subdivision, GPU, real-time, rendering

## 1 Introduction

*Motivation.* Modern game engines provide support for large-scale components such as terrains, water systems, or planetary globes as shown in Figure 1. Due to their intrinsic size and resolution, traditional level-of-detail approaches like Nanite [Karis et al. 2021] are not applicable as the amounts of geometry involved are simply too large[1]. Instead, these large-scale components rely on more ad-hoc solutions that tend to fail in some respect. For instance:

- clipmaps [Asirvatham and Hoppe 2005; Losasso and Hoppe 2004] coupled with static meshes for large vistas [Epic 2023] are hard to balance because they depend on the viewing altitude,
- projected grids [Hinsinger et al. 2002] work well for animated surfaces but produce artifacts on static geometry during camera flythroughs,
- adaptive grids such as quadtrees are either implemented on the CPU [Etienne 2023] or limited in resolution [Deliot et al. 2021; Dupuy 2020].

In this work, we describe an adaptive meshing scheme that is free from the aforementioned limitations and suitable for managing level-of-detail for the geometry of these specialized components.

*Contributions and Outline.* Our meshing scheme computes adaptive triangulations on the GPU progressively through time. As such, it produces optimized amounts of geometry at any time and from ground to space. In terms of contributions, it represents an improvement of the concurrent binary tree (CBT) method [Dupuy 2020] in two distinct aspects, which we detail in the remainder of this article:

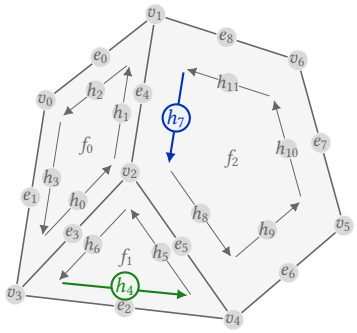
- In Section 2, we describe an algorithm similar to ROAM [Duchaineau et al. 1997] that produces adaptive triangulations for halfedge meshes rather than just squares as done in the original CBT paper.
- In Section 3, we describe a parallel implementation of this sequential algorithm by using a CBT as a memory manager, i.e., in a different way than suggested in the original CBT paper.
- In Section 4, we validate our approach through various rendering results and detailed performance measurements.

Thanks to the way we improve, re-use and combine the ROAM and CBT algorithms, we render planetary surfaces with centimetric detail entirely on mid-range GPUs and using a single representation. Our source code is available online: https://github.com/AnisB/large_cbt.

## 2 Adaptive Triangulation Algorithm

In this section, we describe the adaptive triangulation algorithm that we will implement in parallel in Section 3. We start by providing some background and position our algorithm with respect to related work (Section 2.1). Next, we describe how we initialize our algorithm using halfedge operators (Section 2.2). Finally, we provide each operation necessary to compute adaptive triangulations progressively through time (Section 2.3).

[1] as an example, storing the fully subdivided mesh of Figure 1 would require exa-Bytes of data, which is orders away from what a modern SSD can store




| label | content | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Twin | 6 | 7 | -1 | -1 | -1 | 8 | 0 | 1 | 5 | -1 | -1 | -1 |
| Next | 1 | 2 | 3 | 0 | 5 | 6 | 4 | 8 | 9 | 10 | 11 | 7 |
| Prev | 3 | 0 | 1 | 2 | 6 | 4 | 5 | 11 | 7 | 8 | 9 | 10 |
| Vert | 3 | 2 | 1 | 0 | 3 | 4 | 2 | 1 | 2 | 4 | 5 | 6 |
| Edge | 3 | 4 | 0 | 1 | 2 | 5 | 3 | 4 | 5 | 6 | 7 | 8 |
| Face | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |

Fig. 2. Halfedge mesh representation used for our bisection based subdivision scheme. The blue and green halfedges are highlighted as illustrative examples of a regular and border halfedge, respectively.

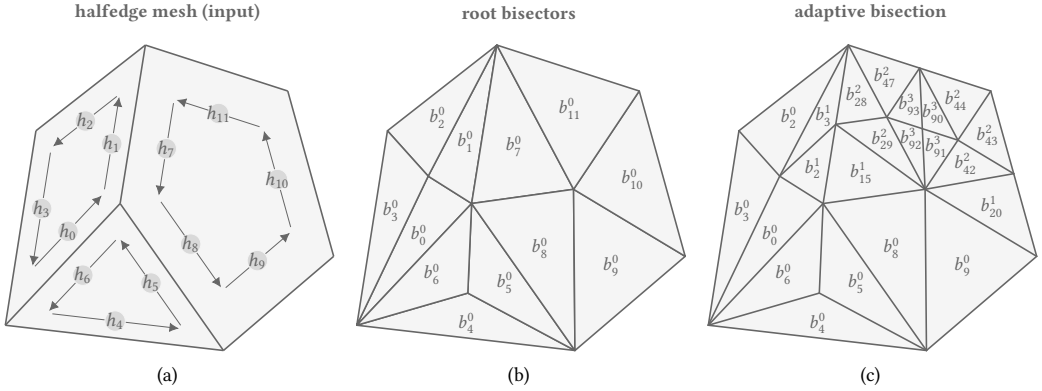



Fig. 3. Overview of our triangulation method.

## 2.1 Background

*Halfedge Meshes as Input.* Our input solely consists of a halfedge mesh. A halfedge mesh [Botsch et al. 2010; Kettner 1999; Weiler 1985] represents a polygon mesh as a set of vertex points and halfedges where each halfedge is characterized by its operators Twin, Next, Prev, Vert, Edge, and Face. Our implementation relies on a generalization of directed edges [Campagna et al. 1998; Dupuy and Vanhoey 2021], which is illustrated in Figure 2 along with all the aforementioned halfedge operators.

*Bisection Based Triangulation.* Our triangulation algorithm is illustrated in Figure 3 and consists in bisecting triangles along one of their edges. Triangle bisection arises frequently in the scientific literature, though under different names. Possible names include, e.g., bisection refinement [Maubach 1995], triangle bintrees [Duchaineau et al. 1997], 4-8 refinement [Velho 2000; Velho and Zorin 2001], diamonds [Hwa et al. 2004; Weiss 2011; Weiss and De Floriani 2011; Yalçın et al. 2011], right-triangulated irregular networks [Evans et al. 2001; Lindstrom et al. 1996], hierarchical simplicial

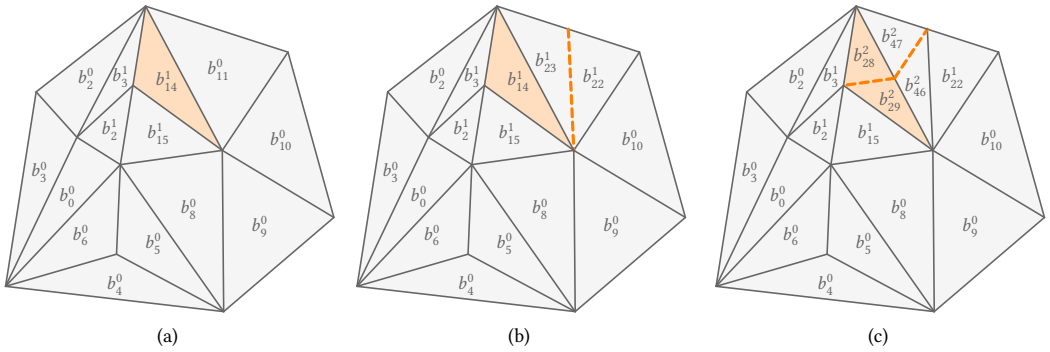



Fig. 4. Refinement over the compatibility chain of bisector $b^1_{14}$ (highlighted in orange).

meshes [Atalay and Mount 2007], newest vertex bisection [Mitchell 2016], and finally longest-edge bisection [Lindstrom and Pascucci 2002; Özturan 1996; Rivara 1984]. The main difficulty with triangle bisection is that it is known to be NP-hard to initialize on arbitrary meshes [Mitchell 2016]. Our contribution in this particular area is to show that the halfedges of the input mesh provide an intrinsic triangulation that provides a valid initialization. This has two benefits. First, it results in an extremely simple algorithm. Second, it preserves the topology of the input mesh (as opposed to existing methods, e.g., [Velho and Zorin 2001]), making it more suitable for animated assets. Aside from our original initialization method, the logic of the algorithm as described in this section is very similar to the ROAM method [Duchaineau et al. 1997].

## 2.2 Initialization: Root Bisectors as Halfedges

We initialize root bisectors at the surface of an input halfedge mesh as shown in Figure 3 (a, b). The properties of these root bisectors are straightforward:

*A Bisector for Each Halfedge.* Each halfedge maps to exactly one root bisector. For instance, the halfedge $h_7$ in Figure 3 (a) maps to the root bisector $b^0_7$ in Figure 3 (b). Note that we refer to a bisector using the notation $b^d_j$, where $d \geq 0$ denotes its subdivision level and $j \geq 0$ its index.

*Root Bisector Vertices.* We retrieve the vertices of each bisectors thanks to the halfedge operators Next and Prev. For instance, the vertices of the root bisector $b^0_7$ in Figure 3 (b) are

$$\begin{aligned} \text{the vertex } v_0 &:= \textsc{Vert}(h_7), \\ \text{the vertex } v_1 &:= \textsc{Vert}(\textsc{Next}(h_7)) = \textsc{Vert}(h_8), \\ \text{and the average } v_2 &:= \frac{1}{5}\left(\textsc{Vert}(h_7) + \textsc{Vert}(h_8) + \cdots + \textsc{Vert}(h_{11})\right). \end{aligned}$$

Algorithm 1 provides pseudocode for computing these vertices.

**Algorithm 1** Root Bisector Vertices

```
1: function RootBisectorVertices(halfedgeID: integer)
2:     nextID ← Next(halfedgeID)
3:     v0 ← Vert(halfedgeID)
4:     v1 ← Vert(nextID)
5:     v2 ← v0
6:     n ← 1
7:     h ← nextID
8:     while h ≠ halfedgeID do
9:         v2 ← v2 + Vert(h)
10:        n ← n + 1
11:        h ← Next(h)
12:    end while
13:    v2 ← v2/n
14:    return [v0, v1, v2]^T
15: end function
```

**Algorithm 2** Bisector Vertices

```
1: function BisectorVertices(b_j^d: bisector)
2:     halfedgeID ← j/2^d                ▷ root bisector index
3:     M ← I                             ▷ identity init.
4:     h ← j
5:     while h ≠ halfedgeID do           ▷ compute subd. matrix
6:         b ← BitwiseAnd(h , 1)
7:         M ← M × M_b                   ▷ see Eq. (1)
8:         h ← h/2
9:     end while
10:    return M × RootBisectorVertices(halfedgeID)
11: end function
```

*Neighboring Bisectors.* We retrieve neighboring bisectors thanks to the halfedge operators Next, Prev, and Twin. For instance, the bisectors adjacent to the bisector $b_7^0$ in Figure 3 (b) are

$$\begin{aligned}\textsc{Next}(h_7) = h_8 &\mapsto b_8^0,\\ \textsc{Prev}(h_7) = h_{11} &\mapsto b_{11}^0,\\ \textsc{Twin}(h_7) = h_1 &\mapsto b_1^0.\end{aligned}$$

### 2.3 Progressive Subdivision Operators

In the following paragraphs, we provide two key operations, namely:

(1) bisector *refinement*, which increases the resolution of the mesh, and
(2) bisector *decimation*, which conversely decreases the resolution of the mesh.

A key property of these operations is that they guarantee conforming topologies at any given time when applied locally (as opposed to globally). Furthermore, they allow for "incremental updates" of an existing triangulation (typically the one from the previous frame) into a slightly more appropriate one in the spirit of ROAM [Duchaineau et al. 1997]. This is done by iterating over existing bisectors and deciding whether to refine, decimate or leave each one of them as is. The key benefit of incremental updates is that they provide an alternative to computing entire triangulations in either bottom-up or top-down fashion. This is particularly important for large-scale game components as they require large subdivision depths.

*Notation.* Refining a bisector $b_j^d$ splits it into two new ones. We denote these new bisectors as

$$\begin{aligned} b_j^d &\mapsto b_{2j}^{d+1} \qquad \text{(first child)}\\ &\mapsto b_{2j+1}^{d+1} \qquad \text{(second child).}\end{aligned}$$

In Figure 4 for instance, the bisector $b_{14}^1$ shown in inset (b) splits into the bisectors $b_{28}^2$ and $b_{29}^2$ shown in inset (c). The reason why we index the children in this way is because we can retrieve their vertices straightforwardly through its subdivision matrix as we show next.

*Bisector Vertices via its Subdivision Matrix.* The bisector refinement rule consists in splitting the bisector into two new ones according to the following refinement matrices (note the difference

between the first two rows):

$$M_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}, \quad \text{and} \quad M_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}. \tag{1}$$

Based on the way we index our bisectors, we retrieve the subdivision matrix associated with any bisector as shown in lines (3–9) of Algorithm 2. In order to get the vertices we simply multiply this matrix by the vertices of the root bisector as shown in line (10) of Algorithm 2.

*Bisector Refinement.* We implement the newest vertex bisection algorithm [Mitchell 2016], which we provide in Algorithm 3. This algorithm ensures that the triangulations produced by the refinement of any bisector results in a conforming triangulation. This is done by propagating the refinement operation through a so-called compatibility chain, which is unique and determined through the recursive calls in Algorithm 3. We illustrate a practical example of such a propagation in Figure 4 for the bisection of the bisector $b_{14}^1$ shown in orange. In order to make this propagation work, each bisector must have pointers to its direct neighbors, and we discuss next how we implement this.

**Algorithm 3** Adaptive Refinement

```
1:  procedure Refine(b_j: bisector)
2:      b_k ← Twin(b_j)
3:      if b_k ≠ null then
4:          if Twin(b_k) ≠ b_j then
5:              Refine(b_k)
6:          end if
7:          Split(b_j, b_k)                    ▷ non-boundary
8:      else
9:          Split(b_j)                         ▷ boundary
10:     end if
11: end procedure
```

**Algorithm 4** Conservative Decimation

```
1:  procedure Decimate(b_{j1}: bisector)
2:      bitValue ← BitwiseAnd(j1, 1)
3:      b_{j2} ← bitValue ? Prev(b_{j1}) : Next(b_{j1})
4:      b_{j3} ← bitValue ? Next(b_{j1}) : Prev(b_{j1})
5:      if ⌊ j1/2 ⌋ = ⌊ j2/2 ⌋ then
6:          if b_{j3} = null then
7:              Merge(b_{j1}, b_{j2})          ▷ boundary
8:          else if ⌊log2 j1⌋ = ⌊log2 j3⌋ then
9:              b_{j4} ← bitValue ? Next(b_{j3}) : Prev(b_{j3})
10:             if ⌊ j3/2 ⌋ = ⌊ j4/2 ⌋ then
11:                 Merge(b_{j1}, b_{j2}, b_{j3}, b_{j4})   ▷ non-boundary
12:             end if
13:         end if
14:     end if
15: end procedure
```

**Algorithm 5** Pointer Refinement

```
1:  procedure RefinePointers(b_j^d)
2:      next ← Next(b_j^d)
3:      prev ← Prev(b_j^d)
4:      if Prev(next) = b_j^d then
5:          Prev(next) ← b_{2j+1}^{d+1}
6:      else
7:          Twin(next) ← b_{2j+1}^{d+1}
8:      end if
9:      if Next(prev) = b_j^d then
10:         Next(prev) ← b_{2j}^{d+1}
11:     else
12:         Twin(prev) ← b_{2j}^{d+1}
13:     end if
14: end procedure
```

**Algorithm 6** Pointer Decimation

```
1:  procedure DecimatePointers(b_{2j}^{d+1}, b_{2j+1}^{d+1})
2:      next ← Twin(b_{2j+1}^{d+1})
3:      prev ← Twin(b_{2j}^{d+1})
4:      if Prev(next) = b_{2j+1}^{d+1} then
5:          Prev(next) ← b_j^d
6:      else
7:          Twin(next) ← b_j^d
8:      end if
9:      if Next(prev) = b_{2j}^{d+1} then
10:         Next(prev) ← b_j^d
11:     else
12:         Twin(prev) ← b_j^d
13:     end if
14: end procedure
```

*Neighbor Refinement Rule.* Bisector refinement operates either:

(1) in isolation at a boundary as shown in Figure 4 (b),

Table 1. Refinement rules for bisector's neighborhood operators.

| children of $b_j^d$ | children of $b_k^d$ (if and only if $b_k^d \neq$ null) |
|---|---|
| $\textsc{Next}(b_j^d) \mapsto \textsc{Next}(b_{2j}^{d+1}) = b_{2j+1}^{d+1}$<br>$\mapsto \textsc{Next}(b_{2j+1}^{d+1}) = \begin{cases} b_{2k}^{d+1} & \text{if } b_k^d \neq \text{null}, \\ \text{null} & \text{otherwise.} \end{cases}$ | $\textsc{Next}(b_k^d) \mapsto \textsc{Next}(b_{2k}^{d+1}) = b_{2k+1}^{d+1}$<br>$\mapsto \textsc{Next}(b_{2k+1}^{d+1}) = b_{2j}^{d+1}$ |
| $\textsc{Prev}(b_j^d) \mapsto \textsc{Prev}(b_{2j}^{d+1}) = \begin{cases} b_{2k+1}^{d+1} & \text{if } b_k^d \neq \text{null}, \\ \text{null} & \text{otherwise.} \end{cases}$<br>$\mapsto \textsc{Prev}(b_{2j+1}^{d+1}) = b_{2j}^{d+1}$ | $\textsc{Prev}(b_k^d) \mapsto \textsc{Prev}(b_{2k}^{d+1}) = b_{2j+1}^{d+1}$<br>$\mapsto \textsc{Prev}(b_{2k+1}^{d+1}) = b_{2k}^{d+1}$ |
| $\textsc{Twin}(b_j^d) \mapsto \textsc{Twin}(b_{2j}^{d+1}) = \textsc{Prev}(b_j^d)$<br>$\mapsto \textsc{Twin}(b_{2j+1}^{d+1}) = \textsc{Next}(b_j^d)$ | $\textsc{Twin}(b_k^d) \mapsto \textsc{Twin}(b_{2k}^{d+1}) = \textsc{Prev}(b_k^d)$<br>$\mapsto \textsc{Twin}(b_{2k+1}^{d+1}) = \textsc{Next}(b_k^d)$ |

(2) in pairs of same subdivision level as shown in Figure 4 (c).

Let $b_j^d$ and $b_k^d = \textsc{Twin}(b_j^d)$ denote the bisectors forming such a pair. If $b_j^d$ is located at a boundary, we have $b_k^d =$ null. The neighbors of the children of $b_j^d$ and $b_k^d$ are given by the rules compiled in Table 1. Note that when a bisector is refined, its neighbors must also be updated to reference the newly created bisectors according to the rules of Algorithm 5.

*Bisector Decimation.* Decimation is the reverse operation of refinement and we could simply implement it as such. In practice though, we rely on a more conservative approach, which avoids the ambiguous case where we decimate entire compatibility chains that carry bisectors under refinement. Our approach thus only halves the number of bisectors in the two following configurations:

(1) we decimate four bisectors of same subdivision depth forming a cycle via their Next and Prev operators into two coarser ones; in Figure 4 for instance, we decimate the four bisectors $b_{28}^2$, $b_{29}^2$, $b_{46}^2$ and $b_{47}^2$ shown in inset (c) into $b_{14}^1$ and $b_{23}^1$ shown in inset (b).
(2) we decimate two border bisectors of same subdivision depth having their Twin point to the null bisector (because of the border) into a single one; in Figure 4 for instance, we decimate the bisectors $b_{22}^1$ and $b_{23}^1$ shown in inset (b) into $b_{11}^0$ shown in inset (a).

The resulting logic is shown in Algorithm 4. Note that when a bisector is decimated, its neighbors must also be updated to reference the newly created bisector accoring to the rules of Algorithm 6.

## 3 GPU Implementation using a CBT

We wish to parallelize the triangulation method from Section 2 over the bisectors already present in the triangulation. As such, each bisector should be able to concurrently refine (see Algorithm 3), decimate (see Algorithm 4), or leave itself untouched. In addition, each bisector should concurrently update neighbor pointer information (see Algorithms 5, 6). In this section, we show how to use a concurrent binary tree (CBT) to fulfill these requirements. We start by recalling details about the CBT data-structure (Section 3.1). We then show how we use CBTs as a memory pool manager and thread scheduler (Section 3.2). Finally, we provide implementation details (Section 3.3).

### 3.1 Background

*Positioning.* CBTs were introduced along with a set of algorithms to compute binary subdivisions over square domains [Dupuy 2020]. As such, we could have re-used these algorithms for the

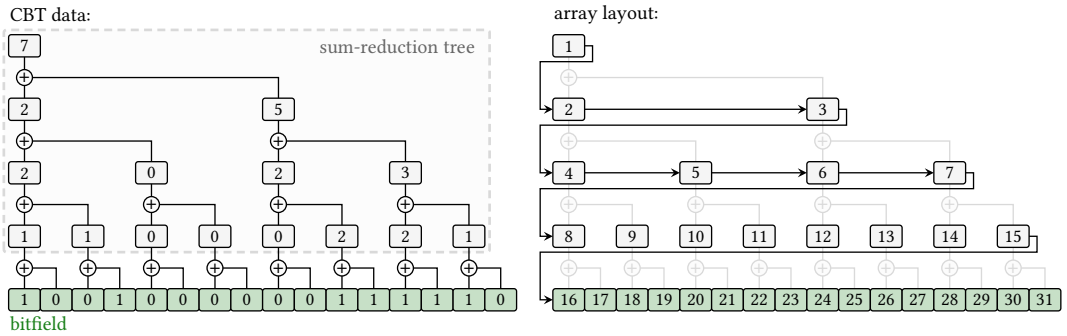



Fig. 5. The CBT's (left) data-structure and (right) contiguous memory layout within an array.

bisector-based scheme from Section 2. Unfortunately though, the original implementation links the maximum subdivision depth of the binary subdivision to that of the CBT. In practice, this restricts the algorithm to subdivisions that are way below what is required for planetary scale rendering (note that we further discuss this issue in Section 4). So rather than re-using the CBT data-structure as is, we modify it so that it can manage a memory pool of bisectors. The memory pool approach is especially relevant for our needs as our bisectors require the same amount of data independently from their subdivision depth. Hence, aside from the memory layout of the CBT, which we recall next, we revisit its usage/implementation entirely as a memory manager.

*Concurrent Binary Tree Data-Structure.* Figure 5 (left) shows an example of a CBT of depth $D = 4$. A CBT [Dupuy 2020] is a full binary tree where:

- the leaf nodes store binary values and form a bitfield,
- the remaining nodes form a sum-reduction tree over these binary values.

Note that, by construction, the root node of a CBT stores the number of bits set to one in the bitfield.

*Data-Layout.* We store a CBT as a binary heap, i.e., an array of size $2^{D+1}$ as shown in Figure 5 (right). This leads to the following properties:

- the children of the node indexed by $k \geq 1$ are located at indexes $2k$ and $2k + 1$,
- the parent of the node indexed by $k \geq 1$ is located at index $k/2$.

This layout avoids the need to store explicit pointers to parent and children within each nodes as they can be computed analytically.

### 3.2 CBT as a Memory Manager

We now provide all the operations required to use a CBT as a memory pool manager. Note that we describe these operations in a generic fashion since they are not tied to our triangulation algorithm (we defer specializations for our triangulation algorithm to the next subsection).

*Initialization.* We allocate a CBT of depth $D \geq 1$ that we use in tandem with a memory pool of capacity $2^D$, i.e., an array of $2^D$ constant-sized memory blocks. These memory blocks must be sized in advance and cannot be re-sized dynamically. In order to track which blocks within the memory pool are available, we pair the $i$-th bit from the CBT's bitfield to the $i$-th block in the memory pool. We thus have a bit value associated to each memory block. By convention, we set bits to one when a block is allocated, and zero when freed/available. Thanks to the CBT's sum-reduction tree, we know how many blocks are allocated by reading the root node. In addition, the CBT's sum-reduction tree

allows scheduling a thread to read from a valid memory block using the following binary search algorithms.

*Binary Search Algorithm.* Given a CBT of depth $D$, we retrieve the index of the $i$-th bit set to one in $O(D)$ as shown in Algorithm 7. For the CBT of Figure 5 for instance, this algorithm returns:

- index 0 for the first bit,
- index 3 for the second bit,
- index 10 for the third bit,
- etc.

Note that, by duality, it is also possible to retrieve the $i$-th bit set to zero with the same complexity, see Algorithm 8. This dual algorithm is important to track available memory blocks.

**Algorithm 7** Find the $i$-th bit set to one

```
1:  function OneToBitID(index: int)
2:      bitID ← 1
3:      while bitID < 2^D do
4:          bitID ← 2 × bitID
5:          if index ≥ CBT[bitID] then
6:              index ← index − CBT[bitID]
7:              bitID ← bitID + 1
8:          end if
9:      end while
10:     return bitID - 2^D
11: end function
```

**Algorithm 8** Find the $i$-th bit set to zero

```
1:  function ZeroToBitID(index: int)
2:      bitID ← 1
3:      c ← 2^(D−1)
4:      while bitID < 2^D do
5:          bitID ← 2 × bitID
6:          if index ≥ (c−CBT[bitID]) then
7:              index ← index − (c−CBT[bitID])
8:              bitID ← bitID + 1
9:          end if
10:         c ← c/2
11:     end while
12:     return bitID - 2^D
13: end function
```

*Usage.* We update the memory pool in parallel through a two-step approach as shown in Algorithm 9. During the first step (see lines 2–16), we iterate over each valid memory block and allow for allocation and de-allocation operations. We detail this step a bit further in the next paragraph. Once this step completed, we simply launch a sum reduction over the CBT's updated bitfield. This completes the update process.

*Block Iteration and Updates.* We parallelize block iteration by scheduling a thread for each memory block. Each thread loads its associated memory block using Algorithm 7. Whenever a thread requests a new memory block, we proceed in the following steps:

(1) we atomically increment an allocation counter that keeps track of the number of allocations,
(2) the original value of this counter tells us which 0-valued bit in the CBT's bitfield should be set to one thanks to Algorithm 8,
(3) once this bit is located, we set it to one and write data to the associated memory block.

As for the case when the thread requests to erase its associated memory block, we simply set the associated bit to zero within the bitfield. After the sum reduction is computed at the end of the update, we will be able to overwrite it safely.

**Algorithm 9** Typical memory pool update using a CBT

```
1: procedure ProcessMemory(cbt, pool)
2:     counter ← 0
3:     blockCount ← cbt[1]
4:     for all threadID ∈ [0, blockCount) do
5:         blockID ← OneToBitID(cbt, threadID)
6:         ...
7:         while mallocRequired do
8:             mallocID ← AtomicAdd(counter, 1)
9:             newBlockID ← ZeroToBitID(cbt, mallocID)
10:            SetBit(cbt, newBlockID, 1)
11:            SetBlock(pool, newBlockID, newData)
12:        end while
13:        ...
14:        if freeRequired then
15:            SetBit(cbt, blockID, 0)
16:        end if
17:    end for
18:    UpdateSumReductionTree(cbt)
19: end procedure
```

### 3.3 GPU Triangulation Pipeline

We provide here implementation details of our CBT-based GPU triangulation algorithm. The objective of this section is two-fold. First, to provide the necessary details suitable for reproducing our implementation. Second, to convince the reader that our parallel implementation is viable.

Table 2. Memory requirements for our implementation.

| attribute name | type | element size |
|---|---|---|
| (input) halfedge buffer | array | ×6 integer |
| (input) vertex buffer | array | ×3 floats |
| bisector pool | $2^D$ array | ×9 integers |
| CBT | $2^{D+1}$ array | ×1 integer |
| allocation counter | scalar | ×1 integer |
| pointer buffer | $2^D$ array | ×2 integers |

*Memory Requirements.* We list all our memory requirements in Table 2. Given a halfedge mesh such as the one shown in Figure 2, we allocate a CBT of depth $D$. The depth $D$ must be large enough to hold all the bisectors of the mesh, i.e., we must have $D \geq \lceil \log_2(H) \rceil$, where $H \geq 3$ denotes the number of halfedges of the input mesh. In Figure 2 for instance, we have $H = 12$ so we must have $D \geq 4$. Note that we typically use $D \in [16, 19]$ in our implementation. This CBT manages a memory pool of capacity $2^D$ bisectors in the spirit of what we described in Section 3.2. We store the following data for each bisector within the memory pool:

- (×3) integers that act as pointers within the memory pool for the neighbors Next, Prev, and Twin of each bisector,
- (×1) integer for the bisector index, i.e., the value $j$ that is required to decode the vertices of the bisector as shown in Algorithm 2.

Note that the width of these integer is important: the neighbor pointers must be sufficiently large to address the entire pool, while the bisector index enforces a maximum subdivision depth. In our implementation, we use 32-bit integers for the neighbor pointers and a 64-bit integer for the bisector index. In addition to these two attributes, we also require data to concurrently split and/or merge each bisector. We store the following additional data for each bisector:

- (×1) integer that encodes a split/merge command as a bit code, which we modify concurrently via atomic bitwise-ORs,
- (×4) integers that store unused memory pool locations where the bisector can write to.

The former integer allows us to make sure splits and/or merges are only evaluated once per bisector. The latter ones are sized based on the fact that we only allow one refinement and/or decimation level per bisector during each incremental update. This means that in the worst case, a bisector will split into four new ones (hence four integers). Such a case happens whenever the bisector belongs to the compatibility chain of two foreign bisectors under refinement. In Figure 4 for instance, splitting bisector $b_{14}^1$ results in splitting $b_{11}^0$ into three new ones due to its location in $b_{14}^1$'s compatibility chain. Finally, we further allocate memory for the following:

- an atomic counter, i.e., an integer, that we increment for each memory allocation,
- a $2^D$ buffer of (×2) integers per element, which caches the results of Algorithms 7 and 8.

We further detail the purpose of each of these attributes in the following paragraphs.

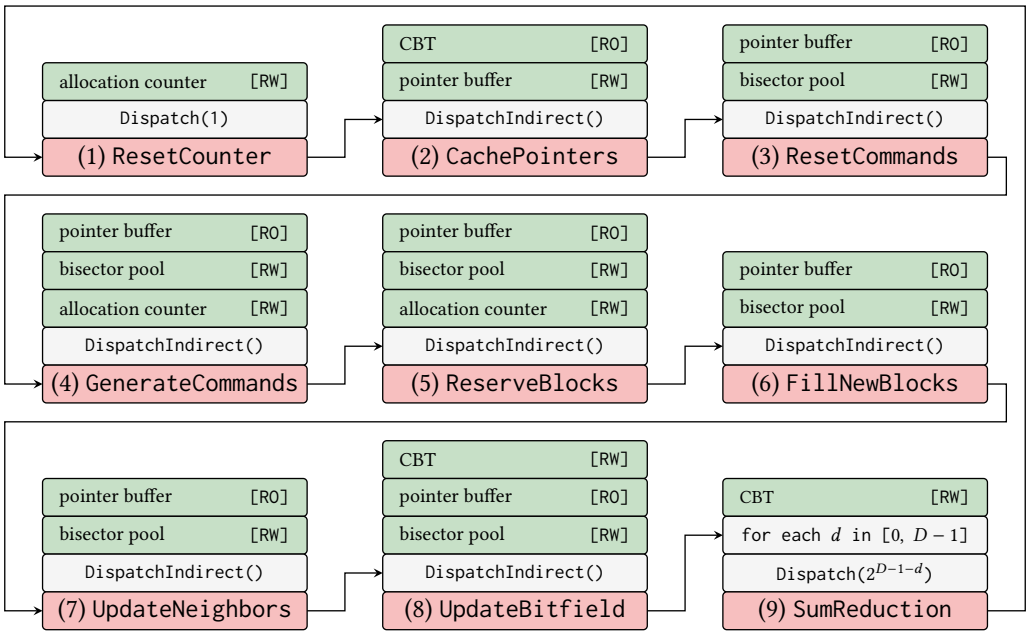



Fig. 6. GPU update loop for our adaptive triangulation algorithm. Memory, CPU commands and GPU kernels are respectively shown in green, gray and red. Each kernel is implicitely followed by a GPU memory barrier.

*Initialization.* We initialize the memory pool with $H$ root bisectors, where we recall that $H \geq 3$ denotes the number of halfedges of the input mesh. We place these bisectors at the first $H$ blocks of the memory pool and set the first $H$ bit of the CBT's bitfield to one (the rest are set to zero). In the memory pool, we uniquely index each bisector as

$$\texttt{bisector.index} = \lceil \log_2 H \rceil + h,$$

where $h \in [0, H-1]$ denotes the index of the mesh's halfedge. As for the neighbors, we simply copy the values of the halfedge buffer

$$\begin{aligned}\texttt{bisector.next} &= \textsc{Next}(h)\\ \texttt{bisector.prev} &= \textsc{Prev}(h)\\ \texttt{bisector.twin} &= \textsc{Twin}(h).\end{aligned}$$

We then compute the sum-reduction tree of the CBT to complete the initialization. The rest of the memory is used during our incremental updates, which we describe next.

*Incremental Update Pipeline.* We update the triangulation incrementally. Each incremental update consists in iterating over each bisector in parallel. In turn, these bisectors can either be refined or decimated once. In our experiments, this approach is enough to handle planetary scales in realtime as we showcase in Section 4. Our GPU update routine is illustrated in Figure 6 and consists of 9 shader kernels, which we detail below:

(1) `ResetCounter`. We simply set the allocation counter to zero. We use this counter to track the number of allocations we need to make to update our triangulation. Note that this kernel requires a single thread invocation, while all subsequent ones require one thread per bisector, whose number we retrieve from the root element of the CBT.

(2) `CachePointers`. We store the result of Algorithm 7 in the first integer of the pointer buffer. This allows us to precompute the index buffer of bisectors located in the memory pool, which we use at the beginning of every subsequent kernel to load the data associated with each bisector (i.e., we use `bisector = MemoryPool[PointerBuffer[threadID]]`). We also store the result of Algorithm 8 in the second integer of the pointer buffer. This precomputes the index buffer of the available blocks in the memory pool, which we use for allocations.

(3) `ResetCommands`. We set the command of each bisector to zero.

(4) `GenerateCommands`. Here we decide whether each bisector should be refined, decimated or kept as is. In our applications, we make our decision by decompressing the vertices of the bisector via Algorithm 2 and making sure its screen-space area matches a user-defined value. We record the decision in the command integer using atomic bitwise-OR operations (by doing so we issue every possible split and merge command only once). We modify the command depending on the decision:

- In case of refinement, we mimic Algorithm 3 by scattering a split command for the active bisector as well as those that belong to the compatibility chain. For the bisectors that belong to the compatibility chain, we need to track which edge of the bisector must be bisected. This is trivial to do thanks to the neighbor operators. Since there are only 3 neighbors per bisector, we reserve 3-bits for the refinement command, each bit corresponding to an edge of the bisector. Note that we condition the execution of this algorithm to the amount of memory available in the CBT. To do so, we atomically increment the allocation counter by the maximum number of allocations. This number depends on the subdivision depth $d$ of the bisector and satisfies

$$\texttt{maxAllocation} = 3d + 4.$$

Thus we only execute the refinement algorithm if the CBT has enough available bits to allocate such a number. In case of overflow, we atomically decrement the allocation counter by the same number and skip refinement for this specific bisector.

- In case of decimation, we mimic Algorithm 4 by first making sure that the bisector belongs to a configuration that can be merged. If it does, we write a merge command to the active bisector. Our merge command records the merge configuration (either triangle or quad as described in Section 2.3) and a flag that tags whether the bisector has the smallest index of the configuration. The latter bit is important to ensure we allocate for the root bisectors only once later in the pipeline. We therefore reserve 3-bits (different from those used for refinement) for the merge command as well. Just as in the case of refinement, we condition the execution of our merge algorithm to the space left in the CBT. Since decimation results in a maximum of two allocations, we increment the allocation counter by two and proceed similarly to the refinement step.
- Finally, if we decide to keep the bisector as is, we leave the command untouched.

At the end of this step, each bisector has a specific command and the allocation counter gives us the number of allocations we have to perform in the CBT.

(5) `ReserveBlocks`. We write to the 4 integers we allocated per bisector to store unused memory pool locations. For each bisector, we read its associated command and determine how many allocations it must make in the memory pool. Due to the way we generate the commands in the previous kernel, a bisector may request splits (at most 4) and/or a merge. When both are required (this happens when a bisector decides to merge and belongs to the compatibility chain of another bisector that requests a split), we ignore the merge command and only apply splits. We then atomically decrement the allocation counter by the number of allocations `nAlloc` required by the bisector, and save the memory locations cached in the pointer buffer in the range [`counter`, `counter + nAlloc - 1`]. In the case of a merge, only the bisector tagged with the minimum index

performs allocations. Such allocations are conditioned to the fact that all relevant bisectors must agree to merge together, i.e., their command must exclusively request a merge. At the end of this step, each bisector has a unique set of memory pool locations where it can safely write new data.

(6) `FillNewBlocks`. Here each bisector writes data to the memory blocks it has allocated. Specifically, we compute and save the index of each newly created bisector. In addition, we set the pointers that inherit from the active bisector according to Table 1. At the end of this step, all bisectors have a valid index and a partial number of valid pointers to their neighbors. We update the remaining pointers in the next step.

(7) `UpdateNeighbors`. Each bisector scatters data to its neighbors. This step is necessary for the data produced by the active bisector's neighbors to properly reference the data produced by the active bisector. Depending on the command, we apply the rules (or compositions of these rules in the case where more than 2 splits are required) of Algorithm 5 and Algorithm 6. At the end of this step, all the bisectors have valid pointers to their neighbors.

(8) `UpdateBitfield`. We update the bitfield of the CBT. Any bisector that has a non-null command must be freed so we set their associated bit to zero. Depending on the command, we set up to 4 bits to one to save the allocations triggered by the bisector.

(9) `SumReduction`. We update the sum-reduction tree of the CBT. After this kernel, we know how many memory blocks are used and, conversely, free in the memory pool. We can thus trigger a new round of incremental update to further refine and/or decimate our input geometry.

## 4 Evaluation and Discussion

In this section, we validate our method and compare its performances against alternative ones.

### 4.1 Test Scenes

*Water Systems.* In Figure 1, we show a water-system running on top of an Earth-sized planet (i.e., roughly 510 million square kilometers) rendered using our triangulation method. We also show a wireframe view segmented as eight successive insets; notice the scale covered by our triangulation and how it adapts to the camera's view frustum to produce equally-sized triangle in screen-space. The planet's base mesh is a regular dodecahedron, which consists of 12 pentagonal faces thus leading to $12 \times 5 = 60$ halfedges and root bisectors. We refine this base mesh over camera-visible areas so that each triangle occupies roughly 49 pixels on-screen. During refinement, we project the triangles' vertices on the surface of a sphere, and further displace them using four layers of choppy waves [Tessendorf et al. 2001]. Each layer is synthesized per-frame on the GPU as a $256 \times 256$ half-float texture using an FFT and displace the vertices both horizontally and vertically with respect to their tangent frame. Once we have updated the triangulation, we export it into a visibility buffer [Burns and Hunt 2013] that we render using the GPU's hardware rasterizer. For shading, we rely on physically-based BRDFs illuminated by a precomputed atmosphere [Bruneton and Neyret 2008]. Thanks to the adaptivety of our triangulation, we are able to render this particular shot using only 64k triangles. We refer the reader to our supplemental video for the animated result.

*Terrain Components.* In Figure 7, we show an elevation-based terrain example over a Moon-sized surface (i.e., roughly 38 million square kilometers) using NASA's moon data, which is available at the following URL: https://svs.gsfc.nasa.gov/cgi-bin/details.cgi?aid=4720. Our elevation data consists of a $5760 \times 2880$ 16-bit displacement map coupled with several octaves of simplex noise. The base mesh and refinement target is the same as the one we described for the ocean system

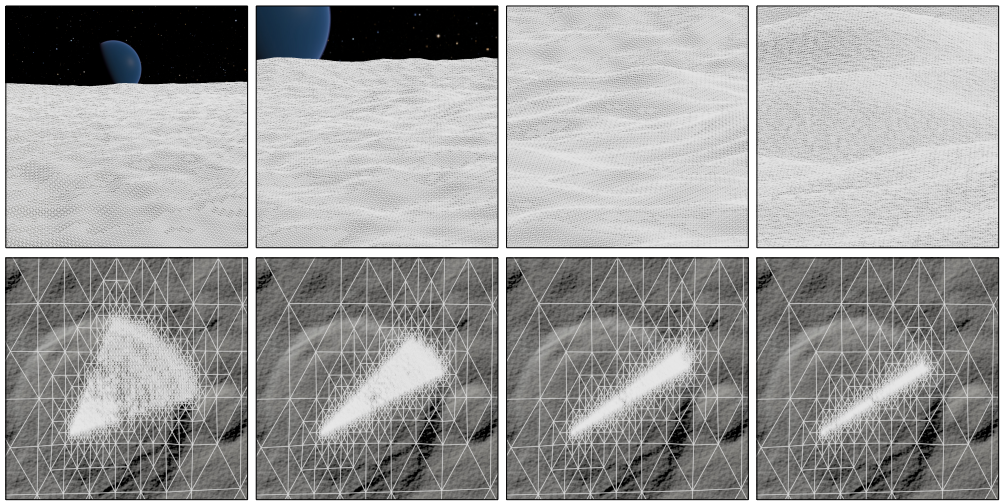


Fig. 7. (top) Frames taken from a zoom-in animation over the Moon terrain. (bottom) Aeral wireframe views.

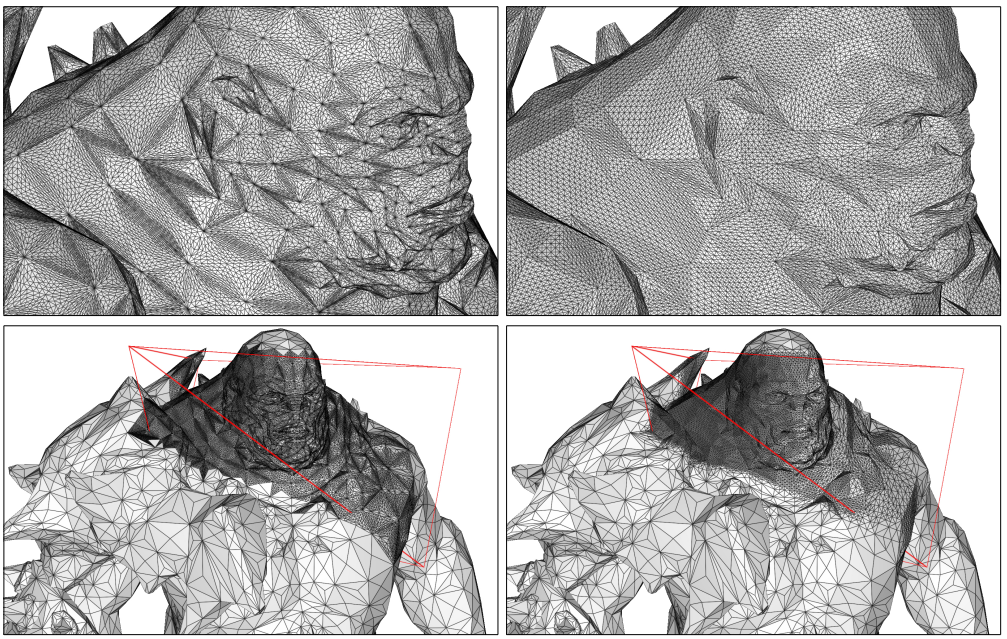


Fig. 8. Comparison between (left) tessellation shaders and (right) our triangulation on a complex mesh.

in the previous paragraph. We also rely on a visibility buffer for rendering. For shading, we use a diffuse BRDF with an albedo that varies according to a $4096 \times 2048$ texture map.

*Arbitrary Asset.* For the sake of completeness, we also evaluated our method against hardware-accelerated tessellation shaders over a complex mesh. In Figure 8, we compare our adaptive triangulation against that of tessellation shaders for a specific view configuration.

Table 3. Performances for various geometries on an AMD 6800 XT GPU. Timings are given in milliseconds.

| test scene ($H$ / $D$) | Figure 1 (60 / 17) | Figure 7 (60 / 17) | Figure 8 (21,399 / 18) |
|---|---|---|---|
| update | 0.1 | 0.084 | 0.085 |
| render | 1.745 | 1.863 | 0.1 |

### 4.2 Performance Measurements

*Absolute Timings.* We benchmarked our test scenes on an AMD 6800 XT, which we chose to mimic a console-level GPU. Since our method runs entirely on the GPU, the CPU does not play a major role in performances. Nevertheless, we mention we use an Intel Core i7-13700K for the sake of completeness. We report our performance measurements for each test scene in Table 3. The update row measures the time taken to compute an incremental update of our triangulation, which comprises all 9 kernels from Figure 6. As demonstrated by the reported numbers, our incremental update is very fast to evaluate, taking less than 0.1ms per frame. Note that update performances are similar between Figure 7 and Figure 8 despite the CBT being twice as big for the latter scene. This is due to the way we implement our `GenerateCommands` kernel, which requires evaluating the world-space position of each bisector's vertices; this evaluation is more expensive for Figure 1 and Figure 7 because we rely on double precision (otherwise discretization artifacts appear as shown in Figure 9). Note that it may be possible to avoid the use of double precision via quantization methods but we have not tried this. As for the rendering row, we sum the timings required to create the visibility buffer and shade the pixels in the case of Figure 1 and Figure 7. For Figure 8 we simply rely on forward rendering.

*Memory Consumption.* Here we provide the memory consumption of our implementation for Figure 1 and Figure 7. We rely on a CBT of depth $D = 17$ and 32-bit integers everywhere except for the bisector index, which is 64-bit wide. This makes a total of 7 MiB of memory (which can be deduced from Table 2) to handle planetary-scale triangulations.

*Performances Against the Original CBT Implementation.* As mentioned in Section 3, the original CBT implementation [Dupuy 2020] couples the maximum depth of the bisectors to that of the CBT, which is set to depth $D = 27$. This has the following implications/limitations:

(1) The root bisectors can be subdivided at most 27 times. While this is enough for moderately-sized terrains, it is way too limited for scenes like those of Figure 1 and Figure 7.
(2) The implementation requires to allocate memory for $2^{27}$ elements despite the fact that most of this memory is not used: in practice the adaptive triangulation produces at most 128k triangles for every viewing conditions we have tried, i.e., over 99% of the memory is wasted.
(3) A sum reduction must be computed over $2^{27}$ elements and is the main bottleneck of the method. On our test hardware, such a computation runs in 0.4ms, which alone is four times slower than our entire update time as reported in Table 3.

In contrast to the original approach, ours decouples the depth of the CBT to that of the subdivision. As a consequence, we reach subdivision depths of up to 64 (as we use 64-bit integers to encode bisector indexes; deeper levels can be achieved with larger integers) with a memory pool of 128k elements and therefore a CBT of depth $D = 17$. Due to the reduced size of the CBT, we also avoid a large sum-reduction compared to the original method. Our novel approach thus produces deeper subdivision levels with less memory while running faster.

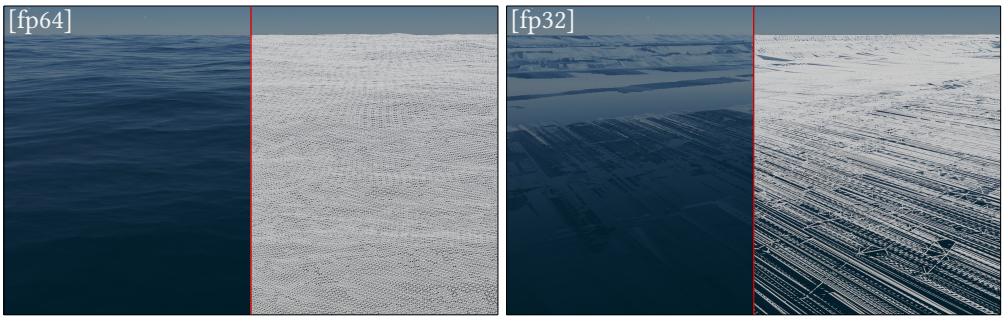



Fig. 9. Impact of double floating point precision on planetary-scale tessellation.

## 5 Conclusion

We introduced an adaptive triangulation method for arbitrary halfedge meshes that runs entirely on the GPU. Our method is primarily geared towards large-scale game components such as terrains, water systems and planets, for which it delivers high performances on console-level GPUs.

## References

Arul Asirvatham and Hugues Hoppe. 2005. Terrain rendering using GPU-based geometry clipmaps. *GPU Gems* 2 (2005), 27–46.

F Betul Atalay and David M Mount. 2007. Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions. *International Journal of Computational Geometry & Applications* 17, 06 (2007), 595–631.

Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon Mesh Processing*. AK Peters / CRC Press. 250 pages. https://hal.inria.fr/inria-00538098

Eric Bruneton and Fabrice Neyret. 2008. Precomputed Atmospheric Scattering. In *Proceedings of the Nineteenth Eurographics Conference on Rendering* (Sarajevo, Bosnia and Herzegovina) *(EGSR '08)*. Eurographics Association, Goslar, DEU, 1079–1086. https://doi.org/10.1111/j.1467-8659.2008.01245.x

Christopher A. Burns and Warren A. Hunt. 2013. The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (12 August 2013), 55–69. http://jcgt.org/published/0002/02/04/

Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. 1998. Directed Edges–A Scalable Representation for Triangle Meshes. *Journal of Graphics Tools* 3, 4 (1998), 1–11. https://doi.org/10.1080/10867651.1998.10487494

Thomas Deliot, Xiaoling Yao, Jonathan Dupuy, and Kees Rijnen. 2021. Experimenting With Concurrent Binary Trees for Large-scale Terrain Rendering. In *ACM SIGGRAPH 2021 Courses*. https://advances.realtimerendering.com/s2021/index.html

Mark Duchaineau, Murray Wolinsky, David E Sigeti, Mark C Miller, Charles Aldrich, and Mark B Mineev-Weinstein. 1997. ROAMing terrain: real-time optimally adapting meshes. In *Proceedings. Visualization'97 (Cat. No. 97CB36155)*. IEEE, 81–88.

Jonathan Dupuy. 2020. Concurrent Binary Trees (with Application to Longest Edge Bisection). *Proc. ACM Comput. Graph. Interact. Tech.* (2020). https://doi.org/10.1145/3406186

J. Dupuy and K. Vanhoey. 2021. A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision. *Computer Graphics Forum* (2021). https://doi.org/10.1111/cgf.14381

Epic. 2023. *Water Meshing System and Surface Rendering in Unreal Engine 5*. https://docs.unrealengine.com/5.0/en-US/water-meshing-system-and-surface-rendering-in-unreal-engine/

Stephan Etienne. 2023. ELarge-Scale Terrain Rendering in Call of Duty. In *ACM SIGGRAPH 2021 Courses*. https://advances.realtimerendering.com/s2023/index.html#CODTerrain

William Evans, David Kirkpatrick, and Gregg Townsend. 2001. Right-triangulated irregular networks. *Algorithmica* 30, 2 (2001), 264–286.

Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. 2002. Interactive Animation of Ocean Waves. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (San Antonio, Texas) *(SCA '02)*. Association for Computing Machinery, New York, NY, USA, 161–166. https://doi.org/10.1145/545261.545288

Lok M Hwa, Mark A Duchaineau, and Kenneth I Joy. 2004. Adaptive 4-8 texture hierarchies. In *IEEE Visualization 2004*. IEEE, 219–226.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A deep dive into Nanite virtualized geometry. In *ACM SIGGRAPH 2021 Courses*. https://advances.realtimerendering.com/s2021/index.html

Lutz Kettner. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry* 13, 1 (1999), 65 – 90. https://doi.org/10.1016/S0925-7721(99)00007-3

Peter Lindstrom, David Koller, William Ribarsky, Larry F Hodges, Nick L Faust, and Gregory Turner. 1996. *Real-time, continuous level of detail rendering of height fields.* Technical Report. Georgia Institute of Technology.

Peter Lindstrom and Valerio Pascucci. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer graphics* 8, 3 (2002), 239–254.

Frank Losasso and Hugues Hoppe. 2004. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. *ACM Trans. Graph.* 23, 3 (aug 2004), 769–776. https://doi.org/10.1145/1015706.1015799

Joseph M Maubach. 1995. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal on Scientific Computing* 16, 1 (1995), 210–227.

William F Mitchell. 2016. 30 years of newest vertex bisection. In *AIP Conference Proceedings*, Vol. 1738. AIP Publishing.

Can Özturan. 1996. *Worst Case Complexity of Parallel Triangular Mesh Refinement by Longest Edge Bisection.* Technical Report. Institute for Computer Applications in Science and Engineering.

M Cecilia Rivara. 1984. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International journal for numerical methods in Engineering* 20, 4 (1984), 745–756.

Jerry Tessendorf et al. 2001. Simulating ocean water. *SIGGRAPH Course notes* (2001), 5.

Luiz Velho. 2000. Semi-regular 4-8 refinement and box spline surfaces. In *Proceedings 13th Brazilian Symposium on Computer Graphics and Image Processing (Cat. No. PR00878)*. IEEE, 131–138.

Luiz Velho and Denis Zorin. 2001. 4–8 Subdivision. *Computer Aided Geometric Design* 18, 5 (2001), 397–427.

K. Weiler. 1985. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Computer Graphics and Applications* 5, 1 (1985), 21–40. https://doi.org/10.1109/MCG.1985.276271

Kenneth Weiss. 2011. *Diamond-based models for scientific visualization.* Ph. D. Dissertation.

Kenneth Weiss and Leila De Floriani. 2011. Simplex and diamond hierarchies: Models and applications. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 2127–2155.

M Adil Yalçın, Kenneth Weiss, and Leila De Floriani. 2011. GPU algorithms for diamond-based multiresolution terrain processing. In *Eurographics Symposium on Parallel Graphics and Visualization*. 10–11.